Отчет по дисциплине «Параллельные вычисления»

Евгений Хандыго, гр. 53501/3

17 апреля 2016 г.

Содержание

1 Постановка задачи

В рамках данной работы необходимо решить задачу вычисления площади фигуры, состоящей из произвольного набора окружностей (возможно пересекающихся) методом *Монте-Карло* (*Monte-Carclo*). Решение задачи должно быть представлено в виде программы на языке `C++` или `Java`. При этом реализация программы должна предусматривать использование соедующих подходов:

- Последовательный.
- Параллельный, с использованием стандартных средств управления потоками выбранного языка программирования.
- Параллельный, с использованием парадигмы `OpenMP` или `MPI`.

Результаты работы необходимо представить в виде статистики замеров времени исполнения различных решений и их сравнения. Также необходимо провести комплексное тестирование разработанных програм для доказания корректности их работы.

2 Введение

Метод Монте-Карло в общем случае применяется для приближенного вычисления интегралов (фактически) произвольной сложности. Суть метода заключается в том, чтобы оценить плотность распределения, график которой в точности равен подинтегральной функции. Таким образом, метод Монте-Карло в некотором роде коррелирует с методом *выборки с отклонением* (*rejective sampling*).

Мы здесь не будем вдаваться в детали данного метода, опишем лишь алгоритм оценки меры произвольной фигуры методом Монте-Карло:

1. Фигура, меру которой необходимо оценить, заключается в другую фигуру, которая:
   - «Достаточно» оптимальна с точки зрения минимальности замыкающей фигуры.
   - Может быть «легко» использована в качестве области определения для генератора равномерно распределнных точек.
2. В замыкающей фигуре генерирутеся «достаточно» большое число n равномерно распределенных точек. Для каждой из точки определяется попала ли она в фигуру, площадь которой необходимо вычислить.
3. Производится подсчет количества точек (m), которые попали в фигуру, площадь которой необходимо вычислить. Тогда искомая величина может быть оценена по следующей формуле:
   $$S * \frac{m}{n},$$
   где S — площадь замыкающей фигуры.

В дополнение к описанному алгоритму заметим, что точность метода Монте-Карло является величиной порядка $\frac{1}{n^2}$, что в свою очередь делает n «достаточно» хорошим при значениях порядка 10^3. Понятно, что в общем случае (например, если оцениваемая фигура сильно разряжена) может потребоваться и гораздо большее количество экспериментов.

3 Детали реализации

В рамках данной работы было принято решения реализовать требуемые программы на языке `Java`. Для реализации третьего варианта решения задачи было решено использовать парадигму `MPI`, поскольку существует его расширения для языка программирования `Java` — `MPJ Express Project` [1]. Данная реализация использует нативную реализацию `MPI` через `Java Native Interface` (`JNI`).

Разработанный в рамках данной работы проект может быть разделен на следующие части:

- Классы, описывающие геометрические фигуры и взаимоотношения меджу ними.
- Классы, реализующие непосредственно задачу оценки фигуры методом Монте-Карло.
- Классы, предназначенные для запуска экспериментов и сбора статистики.
- Тестовые классы.

Ниже на рисунках 1 – 3 представлены диаграммы классов основных частей проекта.

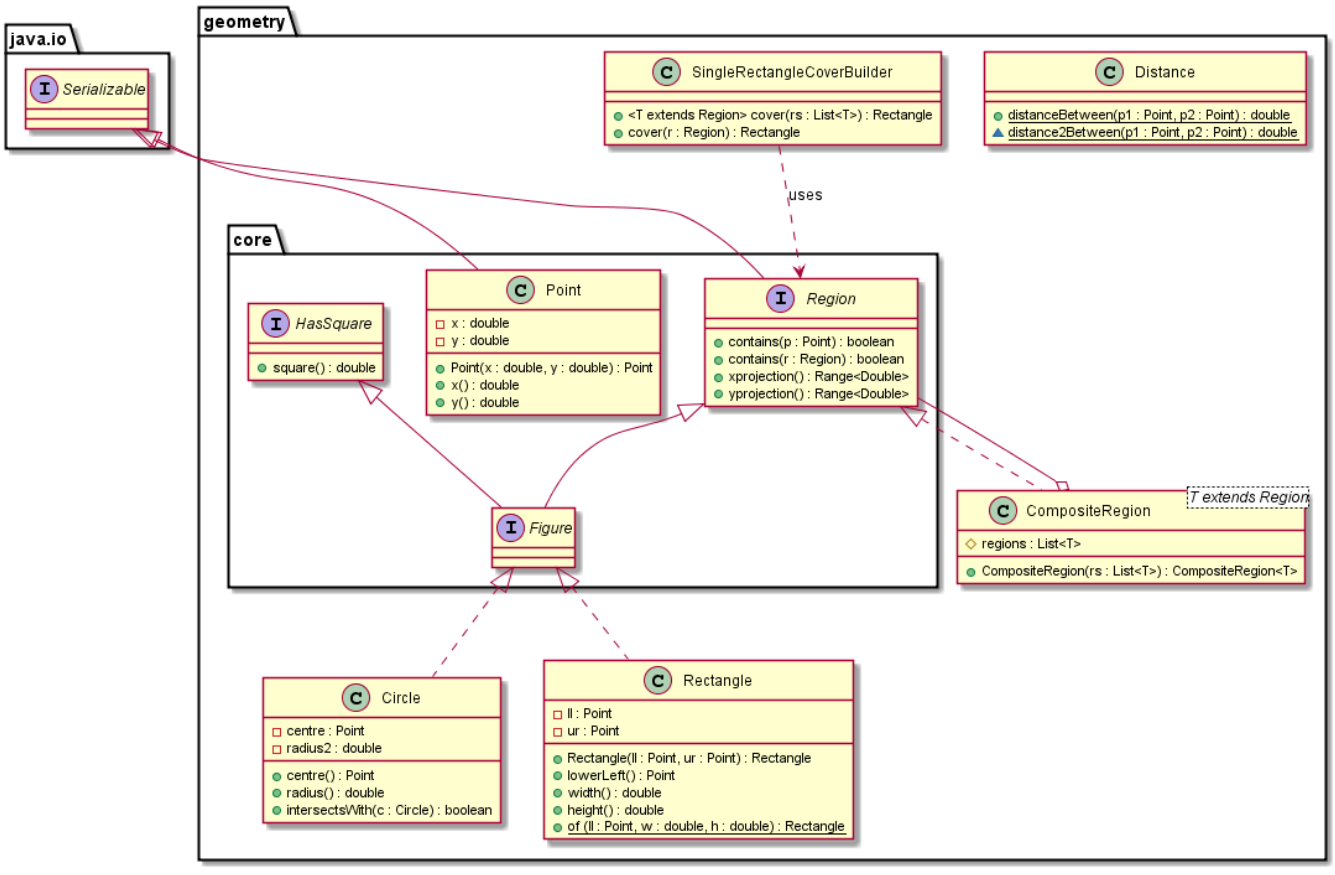

Рис. 1: Диаграмма класов для пакета geometry

[1] http://mpj-express.org/index.html

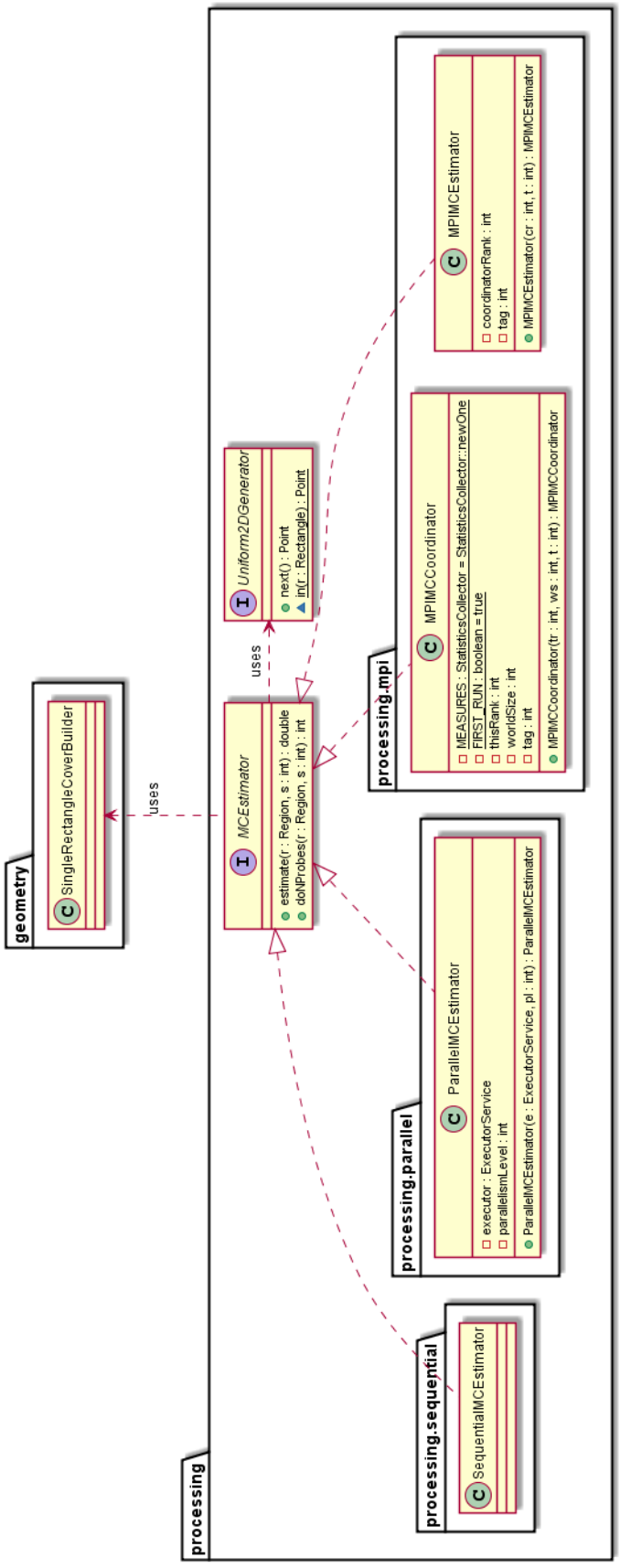

Рис. 2: Диаграмма класов для пакета processing

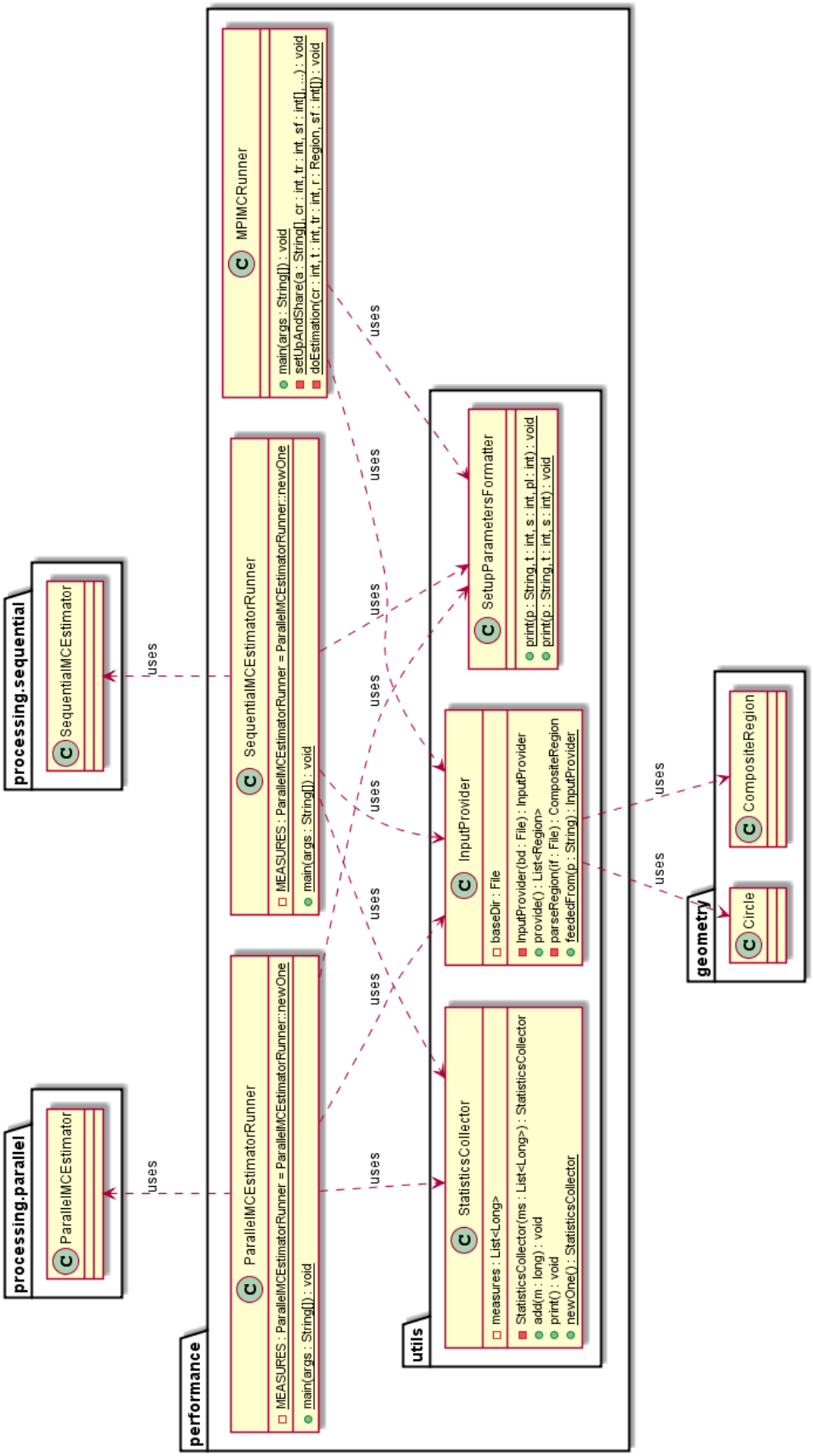

Рис. 3: Диаграмма класов для пакета performance

Ниже в листинге 3.1 приведен код класса `MCEstimator`, реализующего основную функциональность приложения. Остальные классы лишь расширяют его (как показано на рисунке 2), используя функцию `doNProbes` для компонования результатов в случае нескольких потоков.

```
1  package com.ksp.khandygo.processing;
2
3  import com.ksp.khandygo.geometry.Rectangle;
4  import com.ksp.khandygo.geometry.SingleRectangleCoverBuilder;
5  import com.ksp.khandygo.geometry.core.Region;
6
7  public interface MCEstimator {
8
9    double estimate(final Region regionTiEstimate, final int sampleSize);
10
11   default int doNProbes(
12       final Region regionTiEstimate,
13       final int sampleSize) {
14     int successfulProbes = 0;
15     final Rectangle r = new SingleRectangleCoverBuilder()
16         .cover(regionTiEstimate);
17     final Uniform2DGenerator g = Uniform2DGenerator.in(r);
18     for (int i = 0; i < sampleSize; ++i)
19       if (regionTiEstimate.contains(g.next())) ++successfulProbes;
20     return successfulProbes;
21   }
22 }
```

Listing 3.1: Исзодный код класса MCEstimator

4 Эксперименты

4.1 Условия проведения экспериментов

Для реализации поставленной задачи был выбран язык программирования `Java` версии 8. Все эксперименты проводились на вычислительной машине со следующими характеристиками:

- Операционная система Windows 7 Enterprise 64-bit.
- Процессор Intel(R) Core(TM) i5 M 520 с двумя физическими ядрами общей частотой 2.4 гигагерца.
- Объем оперативной памяти 8 гигабайт.

Для запуска экспериментов тестирующая программа собиралась в единый `.jar` файл, включающий в себя все необходимые зависимости. Для гарантирования идентичности усло-

вий проведения экспериментов все эксперименты могут быть запущены с помощью специальных скриптов.

4.2 Методика проведения экспериментов

Эксперименты проводились в два этапа:

1. Генерация исходных данных (единожды для запуска трех различных решений).
2. Запуск каждого вида решения по числу различных экземпляров сгенерированных данных и сбор статистики.

При этом

- Исходные данные представляются в виде файлов в формате `.json`, которые описывают входные данные в стиле POJO (Plain Old Java Objects).
- Каждый файл входных данных описывает фигуру, состоящую из 1000 произвольных окружностей.
- Запуск решений, предполагающих использование многопоточности, производится для количества потоков от 1го до 8ми.
- Замеры времени производятся изнутри самой программы и включают в себя оценку только лишь процесса вычисления.

Ниже представлены параметры установки, верные для всех трех решений.

Параметр	**Значение**
Количество запусков	1000
Количество генерируемых точек	100000

4.3 Результаты экспериментов

Ниже представлены результаты измерений для последовательной программы.

Количество потоков		1
Статистика (мс)	Минимум	27
	Среднее	29.54
	Дисперсия	2.74
	Максимум	99

Ниже представлены результаты измерений для многопоточной программы с использованием стандартных средств языка `Java`.

Количество потоков		1	2	3	4	5	6	7	8
Статистика (мс)	Минимум	28	14	16	14	18	17	16	16
	Среднее	31.47	19.08	20.11	18.09	23.00	20.95	20.33	19.14
	Дисперсия	3.43	5.42	4.60	5.29	5.17	5.11	5.53	8.03
	Максимум	122	117	143	154	164	158	170	253

Ниже представлены результаты измерений для многопоточной программы с использованием парадигмы `MPI` для `Java`.

Количество потоков		1	2	3	4	5	6	7	8
Статистика (мс)	Минимум	30	15	10	8	7	6	5	5
	Среднее	33.49	17.75	16.76	15.70	17.59	19.41	19.88	19.20
	Дисперсия	2.99	3.23	4.40	8.66	18.86	30.72	24.53	26.71
	Максимум	89	50	61	255	409	755	155	169

5 Вывод

В рамках данной работы была разработана программа, позволяющая решить задачу приближенного вычисления площади фигуры, состоящей из произвольного количества окружностей методом Монте-Карло. Программа была реализована с использованием трех различных подходов: последовательного, параллельного, с использованием стандартных средств обеспечения параллелизма в языке программирования `Java`, и с использованием реализации `MPI` для `Java`. Также были проведены эксперименты для замера времени исполнения различных решений. Из результатов экспериментов можно сделать следующие выводы:

- Время работы последовательной программы сопоставимо с временем работы обоих параллельных решений, что может свидетельствовать о корректности постановки экспериментов.
- Минимальное время работы в случае параллельных программ с использованием двух потоков, в два раза ниже минимального времени работы соответствующих программ в один поток. Данный результат полностью соотносится с тем фактом, что для экспериментов была использована машина с двумя физическими ядрами.
- При наращивании количества потоков в параллельных версиях программы, начиная с трех потоков, не наблюдается заметного ускорения по сравнению со случаем двух потоков, что также соотносится с окружением, в котором проводились эксперименты.
- Минимальное время работы при работе с `MPI` ниже, чем минимальное время работы при использовании стандартных потоков языка программирования `Java`, при использовании одинакового количества потоков. В то же время, дисперсия времени исполнения в случае работы с `MPI` выше по сравнению с реализацией на «чистой» `Java` в аналогичных условиях. Здесь можно сделать вывод, что нативная реализация `MPI` в некоторых случаях позволяет выполнить задачу более эффективо, однако, в случае пиковых нагрузок, будет демонстрировать показатели, сопоставимые с показателями стандартных `Java`-потоков или хуже (см случай использования 5и и 6и потоков).